

JavaScript Scope, Closures & Higher Order Functions

(Detailed notes yet beginner friendly)

SCOPE - What Can See What

What is Scope?

Scope = The visibility and accessibility of variables. It answers: "From where can I access this variable?"

Think of scope like **rooms in a house**:

- Variables declared in a room are accessible in that room
- You can look OUT from inner rooms to outer rooms
- You CANNOT look IN from outer rooms to inner rooms

The Three Types of Scope

1. Global Scope (The House Itself)

Variables declared outside any function or block are global.

```
const globalVar = "I'm global";

function someFunction() {
  console.log(globalVar); // ✅ Can access
}

someFunction(); // "I'm global"
console.log(globalVar); // ✅ Can access from anywhere
```

Real-world analogy: Like the air outside – everyone can access it.

2. Function Scope (A Room)

Variables declared inside a function are only accessible inside that function.

```
function myFunction() {  
  const functionVar = "I'm in the function";  
  console.log(functionVar); //  Works  
}  
  
myFunction(); // "I'm in the function"  
console.log(functionVar); //  ReferenceError: functionVar is not  
defined
```

Key point : **var**, **let**, and **const** are all function scoped (but **let/const** are also block-scoped).

3. Block Scope (A Closet in a Room)

Variables declared with **let** or **const** inside **{ }** are block-scoped.

```
if (true) {  
  let blockVar = "I'm in a block";  
  const alsoBlockVar = "Me too";  
  var notBlockScoped = "I'm different!";  
  
  console.log(blockVar); //  Works  
}  
  
console.log(blockVar); //  ReferenceError  
console.log(alsoBlockVar); //  ReferenceError  
console.log(notBlockScoped); //  Works! (var ignores block scope)
```

Important : **var** is not blocked-scope, only function-scoped!

```
function testVar() {  
  if (true) {  
    var x = 10;  
  }  
  console.log(x); //  10 - var leaks out of block!  
}  
  
function testLet() {  
  if (true) {  
    let y = 10;  
  }  
  console.log(y); //  ReferenceError - let respects block scope  
}
```

Lexical scope = Scope is determined by where you write the code, not where you call it.

```
const name = "Global";

function outer() {
  const name = "Outer";

  function inner() {
    const name = "Inner";
    console.log(name); // Which "name" will this print?
  }

  inner();
}

outer(); // "Inner"
```

Why "Inner"? JavaScript looks for variables in this order:

1. **Current scope** (inner function) → Found name = "Inner" ✓
2. If not found, check **outer scope** (outer function)
3. If not found, check **global scope**
4. If still not found → ReferenceError

This is called the Scope Chain.

The Scope Chain in Action

```
const level1 = "Global";

function outer() {
  const level2 = "Outer";

  function middle() {
    const level3 = "Middle";

    function inner() {
      const level4 = "Inner";

      // Can access ALL outer scopes!
      console.log(level4); // "Inner"
      console.log(level3); // "Middle"
      console.log(level2); // "Outer"
      console.log(level1); // "Global"
    }

    inner();
  }

  middle();
}

outer();
```

Visual representation of scope chain:

```
inner() scope
  ↓ (can look up)
middle() scope
  ↓ (can look up)
outer() scope
  ↓ (can look up)
Global scope
```

But you CANNOT look down:

```
function outer() {
  function inner() {
    const secret = "Hidden";
  }

  inner();
  console.log(secret); // ✗ ReferenceError - can't look INTO
  inner function
}
```

CLOSURES - Functions Remember Their Birthplace

What is a Closure?

Closure = A function that remembers variables from its outer scope even after the outer function has finished executing.

This is THE most important concept for understanding JavaScript!

The Basic Closure Example

```
function outer() {  
  const message = "Hello";  
  
  function inner() {  
    console.log(message); // Accesses outer's variable  
  }  
  
  return inner; // Return the function  
}  
  
const myFunction = outer(); // outer() finishes executing  
myFunction(); // "Hello" - but how does it still remember  
"message"?
```


What just happened?

1. **outer()** runs and creates **message**
2. **inner()** is defined INSIDE **outer()** - it "closes over" **message**
3. **outer()** returns **inner** and finishes
4. Normally, **message** would be garbage collected... BUT
5. **inner** still has a reference to **message** - this is a closure!
6. When we call **myFunction()** (which is **inner**), it still remembers **message**

Why Closures Exist

Without closures : Functions would only access their own variables and globals.

With closures: Functions can "carry" their environment with them!

```
function createCounter() {  
  let count = 0; // Private variable  
  
  return function() {  
    count++; // Accesses outer variable  
    return count;  
  };  
}  
  
const counter = createCounter();  
  
console.log(counter()); // 1  
console.log(counter()); // 2  
console.log(counter()); // 3  
  
// "count" is NEVER directly accessible!  
console.log(count); // ✗ ReferenceError
```

Key insight:

count lives on even though **createCounter()** finished! The returned function "closes over" it.

Closures - Real World Example

JS

Real-World Example 1: Private Variables

Closures let you create truly private variables!

```
function createBankAccount(initialBalance) {  
  let balance = initialBalance; // PRIVATE - can't be accessed directly  
  
  return {  
    deposit: function(amount) {  
      balance += amount;  
      return balance;  
    },  
  
    withdraw: function(amount) {  
      if (amount > balance) {  
        return "Insufficient funds";  
      }  
      balance -= amount;  
      return balance;  
    },  
  
    getBalance: function() {  
      return balance;  
    }  
  };  
}  
  
const myAccount = createBankAccount(100);  
  
console.log(myAccount.getBalance()); // 100  
myAccount.deposit(50); // 150  
myAccount.withdraw(30); // 120  
  
// Can't directly access or modify balance!  
console.log(myAccount.balance); // undefined  
myAccount.balance = 9999999; // Doesn't work!  
console.log(myAccount.getBalance()); // 120 - still protected
```

Why this works: All three methods (`deposit`, `withdraw`, `getBalance`) are closures that remember the `balance` variable!

The Simplest Definition

A higher-order function is a function that either:

1. Takes a function as an argument, OR
2. Returns a function as a result

That's it!

Why "Higher-Order"?

Think of it like hierarchy:

- Regular values: numbers, strings, booleans
- First-order functions: functions that work with regular values
- Higher-order functions: functions that work with other functions

In JavaScript, functions are **first-class citizens** - they can be treated like any other value (passed around, returned, stored in variables).

Type 1: Functions That Take Functions as Arguments

Example 1: Array Methods

```
const numbers = [1, 2, 3, 4, 5];

// map is a higher-order function// It takes a function as an
argument
const doubled = numbers.map(function(num) {
  return num * 2;
});

console.log(doubled); // [2, 4, 6, 8, 10]
```

Why is map higher-order? Because it accepts a function (`function(num) { return num * 2 }`) as a parameter.

Example 2: Custom Higher-Order Function

```
// repeat is a higher-order function
function repeat(n, action) {
  for (let i = 0; i < n; i++) {
    action(i);
  }
}

// Using it:
repeat(3, function(i) {
  console.log("Iteration " + i);
});

// Output:// Iteration 0// Iteration 1// Iteration 2
```